Carnegie Mellon
Software Engineering Institute

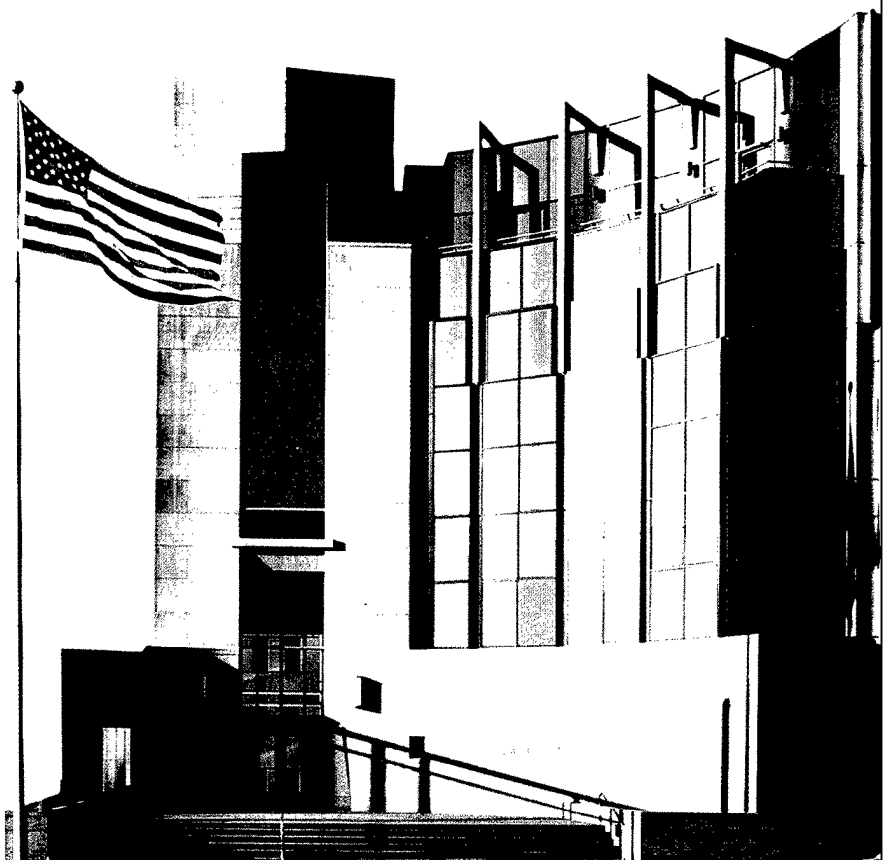
K-BACEE: A Knowledge-Based Automated Component Ensemble Evaluation Tool

Robert C. Seacord
David Mundie
Somjai Boonsiri

December 2000

COTS-Based Systems

Technical Note
CMU/SEI-2000-TN-015



Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of "Don't ask, don't tell, don't pursue" excludes openly gay, lesbian and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Obtain general information about Carnegie Mellon University by calling (412) 268-2000.

K-BACEE: A Knowledge-Based Automated Component Ensemble Evaluation Tool

Robert C. Seacord
David Mundie
Somjai Boonsiri

December 2000

COTS-Based Systems

Technical Note
CMU/SEI-2000-TN-015

Contents

Abstract v

1 Introduction 1

2 System Integration Using COTS 2

3 Architecture 4

4 Implementation 6
- 4.1 Component Specification 6
- 4.2 System Requirements Specification 8
- 4.3 Component Identification 9
- 4.4 Generating Ensembles 9
- 4.5 Evaluating Ensembles 10

5 Conclusions 13

List of Figures

Figure 1: System Requirements Specification 2

Figure 2: Component Selection 3

Figure 3: K-BACEE Architecture 4

Figure 4: Component Specification DTD 6

Figure 5: Tree Representation of a Sample XML Document 6

Figure 6: Use of Primary Keys 7

Figure 7: ILOG JRules Rule Structure 10

Figure 8: Sample Integration Rule 11

Abstract

Component reuse suffers from the inability of system integrators to effectively identify ensembles of compatible software components that can be easily integrated into a system. To address this problem, we have developed an automated process for identifying component ensembles that satisfy a system requirements specification; and for ranking these ensembles based on a knowledge base of system integration rules.

1 Introduction

Enterprise systems, as well as other types of software systems, are increasingly being developed from COTS (commercial off-the-shelf) components. While a COTS-based systems approach is often considered essential to building highly functional, competitive systems, it also poses many challenges.

Foremost among these problems is the evaluation and selection of components. To effectively reuse components, we need to improve the manner in which ensembles of compatible components are identified.

In his ACM Computing Surveys paper on software reuse, Charles Krueger contrasted software reuse technologies based on four criteria: abstraction, selection, specialization, and integration [Krueger 92]. Although it has been clear for some time that integration is a key to software reuse, little has been done to quantify the integrability of components.

In this paper we describe an automated approach to evaluating ensembles of components within the context of a system requirements specification.

2 System Integration Using COTS

COTS-based system development requires the simultaneous survey of the marketplace (including available COTS products and applicable standards), the system context (including requirements, cost, schedule, business processes) and the existing system architecture [Hansen 00]. The system context may also be constrained by corporate standards and policies and the system architecture constrained by existing legacy systems. A component-based software engineering process model that incorporates these factors is described by Ning [Ning 96].

Using the information gleaned in this survey, the system integrator creates an initial system requirements specification (SRS). The SRS describes constraints on the system. In the case of COTS-based systems, the SRS also identifies functional areas that may be satisfied by commercially available components.

The development of the SRS is an iterative process. Successive evaluation of candidate components will cause the SRS to evolve. In fact, it is important not to spend too much time refining the SRS early in the process as this often results in a rigid set of requirements that limits the degree to which COTS products can be successfully integrated.

Constraints:
Language: Java
GUI: Web Browser
Components:
C1: XML DB
C2: rules engine
C3: XML/Java converter

Figure 1: System Requirements Specification

The SRS provides the initial context for evaluating components. Figure 1 shows an SRS for a hypothetical system. This SRS defines constraints, possibly driven by corporate policy decisions, concerning the implementation language and GUI mechanism. The specification also includes a functional description of components required by the system. These components may be purchased or custom developed, but it is incumbent on the system integrator to rule out the use of commercially available components before resorting to custom development.

Once the SRS is defined, the system integrator can identify components that meet the documented functional requirements and systems constraints. Currently, the identification and evaluation of commercial components is largely a manual task. In Figure 2, we show candidate components that provide the functionality required by our example system.

Component **x1**, **x2** and **x3** satisfy the functional requirements for the XML dB, component **y1** satisfies the functional requirements for a rules engine; **z1**, **z2** and **z3** satisfy the functional requirements for an XML/Java converter.

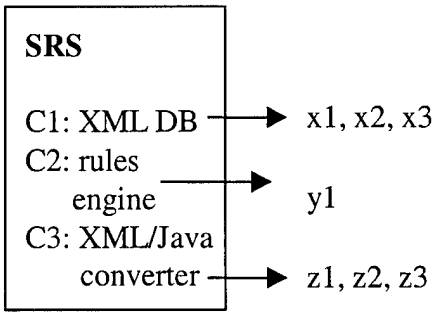


Figure 2: Component Selection

It is insufficient to evaluate each of the individual software components in isolation; this does not guarantee that the selected components are mutually compatible. The system integrator must discover an *ensemble*—a collection of compatible components—that satisfies the functional requirements. When multiple components are found that match the requirements, however, the number of possible ensembles expands, based on the factor of the cardinality of each set of qualified components, according to the formula

$$\prod_{s \in A} \#s$$

where A is the set of component sets. This situation deteriorates further when you consider that there may be multiple versions of each component.

Evaluating all possible component ensembles by hand is, of course, prohibitively expensive. In practice, system integrators must rely on experience to select components with a high degree of compatibility. The result is that the component space is largely unexplored, and the possibility that an optimal component ensemble has been selected is low.

Our proposed solution to this problem is to automate at least part of the process by which component ensembles consisting of compatible components are identified. It is hoped that, by automating part of this process, a savings in evaluation and development costs can be achieved and that a greater percentage of the component space can be considered. We implemented a prototype system to test our solution called K-BACEE (pronounced *kay-base*) for Knowledge-Based Automated Component Ensemble Evaluation.

3 Architecture

Figure 3 illustrates the K-BACEE architecture, consisting of a searchable repository of component specifications, integration rules, query server, and component ensemble evaluator. Users of the system provide an SRS that defines the functional requirements of the components as well as any overriding constraints on how the components are integrated.

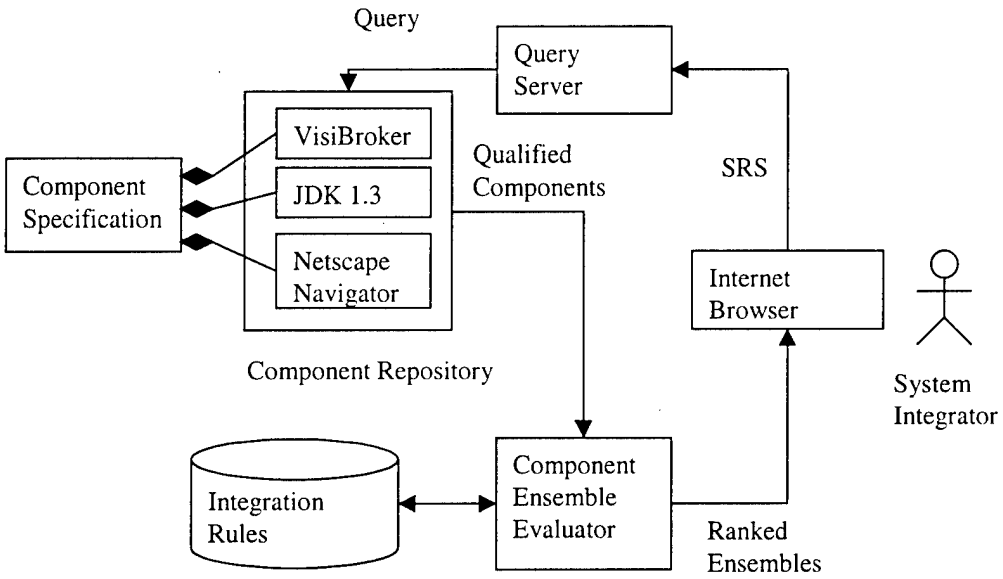


Figure 3: K-BACEE Architecture

The requirements specification is converted into a series of queries on the component repository.[1] Components that match the requirements specified in the SRS are grouped into ensembles.

Components in each ensemble are evaluated for compatibility based on the value of attributes in their component specification and a repository of software engineering integration rules. The attributes define characteristics that impact compatibility with other components, for example, the protocols supported by the component. Integration rules define how attributes affect component integration. These rules identify both those attribute combinations that simplify—and those that complicate—system integration.

The integration rule database may be extended by both system integrators and component vendors using a separate interface not shown in Figure 3. Integration rules typically reflect

[1] The component repository does not contain the actual components (as might be suggested by the name) but component specifications describing the components.

known compatibilities and incompatibilities between products. The discovery and refinement of these rules is a normal part of the system integration process, making it necessary to support processes to all new integration rules to the database as well as modify or delete existing rules.

Product ensembles, ranked according to compatibility, are returned to the user for further evaluation.

4 Implementation

To verify the feasibility of K-BACEE, we implemented a prototype. The implementation of each component of the K-BACEE architecture is described in the following sections.

4.1 Component Specification

XML (eXtensible Markup Language) is used in K-BACEE to represent both the component and system requirement specifications. XML is a World Wide Web Consortium (W3C) recommendation that has become universally accepted as the standard for document interchange [Bray 98]. XML is well suited for this application as it provides a formal language for mapping attributes to values and is fully extensible [Mundie 97, Cover 00]. Figure 4 shows the Document Type Definition (DTD) for a component specification.

```
<?xml encoding="UTF-8"?>
<!ELEMENT components (component+)>
<!ELEMENT component (general_info, protocol, security)>
<!ATTLIST component id ID #REQUIRED>
<!ELEMENT general_info (name, version, vendor, platform,
   categories, function+, framework, language, domain, keywords)>
<!ELEMENT protocol (name, version, provider, protocol)>
<!ATTLIST protocol credential CDATA #REQUIRED>
<!ELEMENT security (confidentiality?, authentication?, non-repudiation?)>
```

Figure 4: Component Specification DTD

The component specification contains general information, component interface information, and a component ID. The attributes included in the component specification borrow from Poulin [Poulin 95]. A tree representation of an XML component specification is shown in Figure 5.

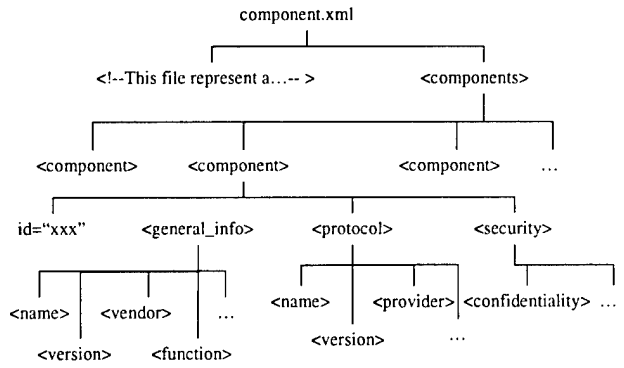


Figure 5: Tree Representation of a Sample XML Document

The general information includes the product name, version, vendor, platform and functionality and then information about the system interfaces. Functionality may be described informally using key terms or with a formal specification language such as the one described by Mili, Mili and Mittermeir [Mili 97]. Vendor information is used to facilitate locating additional information about the component.

Component interfaces documented in the component specification are not simply the signatures of method or function calls in the component, but include non-functional properties such as performance, accuracy, availability, latency, and security This expanded use of the term *interfaces* is described by Bachmann, et. al. in their treatise on component-based software engineering [Bachmann 00].

In addition to general information and component interface information we assign each component a unique identifier.

Component identifiers can be referenced from other component specifications—allowing a component's interface to be defined in terms of another component. For example, a component specification may claim that a component uses CORBA (Common Object Request Broker Architecture) as an integration mechanism and that it supports version 2.0 of IIOP (Internet Inter-ORB Protocol). However, differences in implementation, such as incompatibilities in the naming service may make it more difficult to integrate a component implemented using Inprise VisiBroker with one developed using IONA Orbix [Seacord 99]. Figure 6 shows a component named "Com Pro"(component 432) that provides an IIOP interface.

```
<component id="101">
   <general _info>
     <name>VisiBroker</name>
     <version>3.2</version>
     <vendor>
          <name>Inprises</name>
     </vendor>
   <general_info>
   </protocol credential="provider">
     <name>IIOP</name>
     <version>2.0</version>
   </protocol>
</component>

<component id="432">
   <general_info>
      <name>Com Pro</name>
      <version>1.0</verson>
   </general-info>
   <protocol credential="client">
        <name>IIOP</name>
        <version>2.0</version>
        <provider idref=101">
   </protocol>
</component>
```

Figure 6: Use of Primary Keys

The component specification identifies component 101 (Inprise VisiBroker in this example) as the provider of the software used by the component to support the protocol. Including this information in the component specification provides visibility into the internal design of the component. This information is normally unavailable to component consumers, but provides valuable insight into the implementation of a component. These relationships can be used, for example, to determine a component's suitability for integration into an ensemble.

Prieto-Diaz and Freeman suggest that the characterization of a software component's functionality and its environment suffice for classification [Prieto 87]. We include information about functionality and environment in our component specification but add additional information regarding component characteristics that may affect integration with other components. Gorman states that the inclusion of additional attributes may improve the effectiveness of component search [Gorman 99]. It is important to justify the inclusion of these attributes in the component specification and not just add them extemporaneously. Poulin suggests that collecting large amounts of metadata to help retrieve components wastes time and money, and makes the library both difficult to contribute to and difficult to retrieve from [Poulin 99]. It is therefore necessary to limit the selection of attributes to those that have a significant impact on the suitability of a component for integration. The availability of language bindings, for example, is an important attribute that greatly influences the degree of difficulty involved in integrating a component.

4.2 System Requirements Specification

The system requirements specification is also represented in XML. As used in K-BACEE, the SRS consists of one or more component specifications and a set of system constraints. In an actual system this SRS may in fact be simply one element of a larger artifact.

System constraints use the same collection of attributes as individual components; so, for example, Java may be specified as the language of choice for the system.

Component specifications in the SRS are normally sparsely populated—often the functional requirements alone are specified. This is because elements in the SRS are treated as absolute constraints on the system (they are, after all, requirements). For example, if the platform is specified as WinNT, only components that are available on this platform are identified. Attributes specified in the SRS for individual components similarly limit component candidates to those that match the requirement.

This produces a tension between search constraints and search results and, correspondingly, between requirements and available components. As search constraints are relaxed, additional but less-qualified components will be identified. As additional constraints are added, a smaller group of better-qualified components will be identified.

The ability to modify requirements and re-execute a search is a major benefit of automating this process—allowing system integrators to adjust system requirements to market realities in real time.

4.3 Component Identification

K-BACEE uses the SRS to identify an initial working set of qualified components. To identify all the components in the component repository that satisfy the SRS, we extract the set of constraints from the SRS and transform them into XML Query Language (XQL) [Robie 99] queries. XQL is one of several XML query language proposals; however, it enjoys the support of several commercial products.

Once constructed, XQL queries are run against the component repository to identify a set of candidate components. For example, the SRS may specify a functional requirement for a spell checker component as shown below:

```
<function>spell checker</function>
```

This would be extracted as an XQL query as follows:

```
"//component/general_info/function[spell checker]"
```

Running this query against the component repository generates a working set of components that implement this functionality.

4.4 Generating Ensembles

Once a working set of components has been identified, they must be grouped into possible ensembles. Unfortunately, this is a combinatoric problem, and represents a serious problem for our approach since the number of potential ensembles grows exponentially with the number of component candidates. Examining every possible combination for large numbers of components quickly becomes impractical.

There are several possible solutions to this problem. We decided to use statistical sampling to identify possible ensembles for evaluation. If the data set is small, the statistical approach will eventually select all of them anyway, but if the data space is large, statistical sampling is an effective way of finding a satisfactory ensemble. It is true that the optimal ensemble may be "just over the horizon" at the point the sampling is interrupted, but the chances of that happening can be reduced to any desired level simply by extending the time devoted to evaluation.

We also considered two other approaches. The first was to ask the user to place further constraints on the components, so as to reduce the number of candidates. We rejected this because, even if we did ask the users to further constrain the problem, we wanted to aid them

by showing them sample evaluations of the unconstrained ensembles, so that they could make intelligent choices about which constraints to add.

The second alternative we considered was to provide an interactive tool that would let the integrator explore the ensemble space dynamically. For example, we might generate a random ensemble and let the integrator hold all but one of the components fixed, and evaluate all the ensembles obtained by varying the last component.

Since both of these alternatives required implementing the statistical approach first, and since we believe that the statistical approach may suffice on its own, we decided to limit the prototype to this approach and defer evaluation of these alternatives.

4.5 Evaluating Ensembles

The final component of K-BACEE is the ensemble evaluator that assesses the degree to which components in a candidate ensemble are compatible with one another.

Despite Tim Berners-Lee's vision of the "semantic web" as a place where intelligent agents reason on our behalf about an immense universal ocean of marked-up, semantically rich data, relatively little has been done to produce practical XML-based reasoning systems [Berners-Lee 99, Ogbuji 00]. Most of the existing approaches express the reasoning rules in XML, but the inference engines that can process rules in this form are still experimental. We opted instead to use ILOG JRules—a commercial expert system designed to work with Java. The use of a commercial expert system provides a powerful and intuitive way to capture the domain knowledge and allowed us to complete the prototype with a minimum of effort.

JRules is an object-oriented rule-based programming language. A JRules application consists of a set of rules and a collection of objects. Each rule is composed of a header, a condition part, and an action part. A rule is processed in the following manner: given a set of conditions, each condition with its variables is matched to an object with its field values. A condition may include tests on variable values. If all the conditions are matched (that is, the variables return field values that fulfill the tests) the action part may be executed. The set of actions may include simple operations such as printing text, as well as complex operations such as invoking methods on objects or modifying objects used by the rules. The ILOG JRules structure is shown in Figure 7.

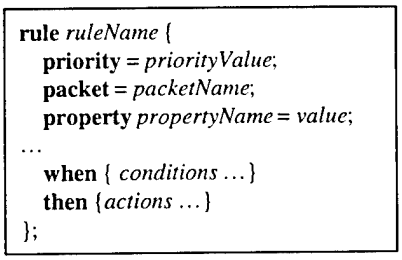


Figure 7: ILOG JRules Rule Structure

Objects in JRules correspond to actual Java objects. To be evaluated by a rule, the object must exist in working memory. Placing an object in working memory is accomplished in JRules through the use of an **ASSERT** statement.

A sample rule for evaluating language compatibility is shown in Figure 8.

```
rule LanguageCompatible3 {
  priority = high;
  when {
    ?c1: Component( lang.equals("JDK1.1"); ?i1:id);
    ?c2: Component( lang.equals("C"); ?i2:id ;
         ?i1 != ?i2);

    ?e: Evaluation();
  }
  then {
    modify ?e {score += 6;}
  }
};
```

Figure 8: Sample Integration Rule

This rule evaluates language compatibility between components written in Java (specifically JDK 1.1) and C programming languages.

The rule consists of two parts: a **when** block and a **then** block. The **when** block defines two *patterns*. Patterns are used in JRules to identify objects. The first pattern in the **when** block matches **Component** object instances that declare a **lang** variable with the assigned value of **"JDK1.1"**. For each such object discovered, **?i1** is assigned the value of the **id** variable defined within the **Component** object. The **Component** object is assigned to the rule variable **?c1.**

The second pattern in the **when** block is executed in a similar fashion. The actions in the **then** block are performed for each combination of **?c1** and **?c1** where **?i1 != ?i2**. This final clause prevents a single component object from being tested for compatibility with itself.

In the above example, if component **?c1** is implemented using JDK 1.1 and component **?c2** is implemented in the C programming language, six points are added to the compatibility score for the ensemble, as a well-defined interface exists between these two languages (the Java Native Interface). Compatibility scores range from –10 (the lowest level) to 10 (the highest level).

Before we could execute these rules, of course, we had to do two important things. First, we had to convert the XML descriptions into Java objects. Second, we had to move these objects into working memory.

One approach for generating Java objects is to use the Java binding of the Document Object Model (DOM). This is a natural application of DOM since it is primarily intended as a document-to-object mapping tool. However, we did not attempt this approach, since we were concerned that the DOM objects did not provide the interface required by JRules. Since the collection of Java objects can be viewed as static, we simply translated a sample set of XML component specifications into Java classes. This translation is mechanical and should be easy to automate.

The problem of moving Java objects into working memory could not be solved as easily, as only those objects returned by the XQL query are moved. We decided to use XSL Transformations (XSLT) [Clark 99] to generate the required ASSERT statements from the XML documents returned by the XQL query. These statements referenced the corresponding Java objects that had been generated statically beforehand.

Once these objects have been moved into working storage, the integration rules can be executed to rank the ensembles. Evaluation scores are accumulated for each ensemble set and the ensembles are presented to the system integrator in order of their rankings.

5 Conclusions

K-BACEE can be thought of as an expert system for system integration. K-BACEE allows system integrators to explore a broader component space than is possible using manual techniques, and quickly eliminate components that are overly difficult to integrate.

In addition to aiding in the evaluation of component ensembles, K-BACEE provides a mechanism for preserving, sharing and re-using hard-won system integration knowledge. This information can then be used by system integrators to identify compatible ensembles of components. The information can be actively expanded by system integrators as they develop additional insights into the rules that govern system integration.

The greatest challenge in K-BACEE is not the feasibility of automating the process, which we feel we have demonstrated, but the ability to collect the data necessary to drive the process. First, it is necessary to populate the component repositories with a sufficient number of component specifications to guarantee that an SRS can be satisfied from the pool of available components. Secondly, it is necessary to identify, through successive rounds of refinement, the attributes that are used to describe each component and the set of system integration rules that are used to compute the compatibility of ensembles.

It is hoped that component brokers, who market third-party components, may initially stand up component search engines based on the K-BACEE model. Component consumers will use this service to discover components that can be easily integrated into their systems and begin to rely on these component brokers. Component producers may then feel compelled to generate component specifications that are compatible with this approach. Optimally, this process will culminate in a standard component repository and component specification format.

References

[Bachmann 00] Bachmann, Felix; Bass, Len; Buhman, Charles; Comella-Dorda, Santiago; Long, Fred; Robert, John; Seacord, Robert; & Wallnau, Kurt. *Technical Concepts of Component-Based Software Engineering, Volume II* (CMU/SEI-2000-TR-008, ADA379930). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 2000. Available WWW <URL: http://www.sei.cmu.edu/publications/documents/00.reports/00tr008.html>.

[Bray 98] Bray, Tim; Paoli, Jean; & Sperberg-McQueen, C.M. *Extensible Markup Language (XML) 1.0.* W3C Recommendation, February 10, 1998.

[Berners-Lee 99] Berners-Lee, Tim. *Weaving The Web.* San Francisco, Ca.: Harper, 1999.

[Boley 00] Boley, H; Decker, S; & Sintek, M. "Tutorial on Knowledge Markup." September 2000. Available WWW <URL: http://www.dfki.uni-kl.de/km/knowmark/>.

[Clark 99] Clark, James. *XSL Transformations (XSLT) Version 1.0.* W3C Recommendation, November 16, 1999. Available WWW <URL: http://www.w3.org/TR/xslt>.

[Cover 00] Cover, Robin. "Literate Programming with SGML and XML." *The XML Cover Pages.* Available WWW<URL: http://www.oasis-open.org/cover/xmlLitProg.html>.

[Gorman 99] Gorman, Michael. "Metadata or Cataloguing? A False Choice." *Journal of Internet Cataloging* 2(1) 1999.

[Hansen 00] Hansen, W.J.; Foreman, J.T.; Carney, D.J.; Forrester, E.C.; Graettinger, C.P.; Peterson, W.C.; & Place, P.R. *Spiral Development–Building the Culture: A Report on the CSE-SEI Workshop* (CMU/SEI-2000-SR-006, ADA382585). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 2000. Available WWW <URL: http://www.sei.cmu.edu /pub/documents/00.reports/pdf/00sr006.pdf>.

[Krueger 92] Krueger, Charles "Software Reuse." *ACM Computing Surveys 24*, 2 (June 1992): 131-183.

[Mili 97] Mili, Rym; Mili, Ali; & Mittermeir, Roland T. "Storing and Retrieving Software Components: A Refinement Based System." *IEEE Transaction on Software Engineering 23,* 7 (July 1997).

[Mundie 97] Mundie, David. "Standardized Data Representations for Software Testing." *Pacific Northwest Conference on Software Quality*, Portland, Maine, October, 1997.

[Ning 96] Ning, Jim Q. "A Component-Based Software Development Model." *Proceedings of 20th International Conference on Computer Software and Applications*, Seoul, Korea: 1996.

[Ogbuji 00] Ogbuji, Uche. "RIL: A Taste of Knowledge." October 2000. Available WWW<URL: http://www.xml.com/pub/2000/10/11/rdf/ril.html>.

[Poulin 99] Poulin, Jeffrey S. "Reuse: Been There, Done That." *Communication of the ACM 42*, 5 (May 1999): 98-100.

[Poulin 95] Poulin, Jeffrey S. & Werkman, Keith J. "Melding Structured Abstracts and the World Wide Web for Retrieval of Reusable Components." *Proceedings of the 17th International Conference, Symposium on Software Reusability on Software Engineering*, Seattle, Wash.: April 23-30, 1995.

[Prieto 87] Prieto-Diaz, R. & Freeman, P. "Classifying Software for Reusability." IEEE Software *4*,1 (January 1987): 6-16.

[Robie 99] Robie, Jonathan. "XQL (XML Query Language)," August 1999. Available WWW <URL: http://www.ibiblio.org/xql/xql-proposal.html>.

[Seacord 99] Seacord, Robert C.; Wallnau, Kurt; John, Robert; Comella-Dorda, Santiago; & Hissam, Scott A. "Custom vs. Off-the-Shelf Architecture." *Proceedings of the 3rd International Enterprise Distributed Object Computing Conference*. Mannheim, Germany, September 27-30, 1999.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	December 2000	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
K-BACEE: A Knowledge-Based Automated Component Ensemble Evaluation Tool	F19628-00-C-0003

6. AUTHOR(S)

Robert C. Seacord, David Mundie, Somjai Boonsiri

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2000-TN-015

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Component reuse suffers from the inability of system integrators to effectively identify ensembles of compatible software components that can be easily integrated into a system. To address this problem, we have developed an automated process for identifying component ensembles that satisfy a system requirements specification and for ranking these ensembles based on a knowledge base of system integration rules.

14. SUBJECT TERMS	15. NUMBER OF PAGES
automated component integration, ensemble evaluation, component specification	24

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102